21장 빌트인 객체

21.1 자바스크립트 객체의 분류

- 표준 빌트인 객체 (standard built-in objects/native objects/global objects)
  - ECMAScript 사양에 정의된 객체.
  - 애플리케이션 전역의 공통 기능을 제공.
  - 자바스크립트 실행 환경과 관계없이 언제나 사용 가능.
  - 전역 객체의 프로퍼티로서 제공되며 전역 변수처럼 언제나 참조 가능.

- 호스트 객체 (host objects)
  - ECMAScript 사양에는 없지만 자바스크립트 실행 환경에서 추가로 제공하는 객체.
  - 브라우저 환경 : 클라이언트 사이드 Web API.
  - Node.js 환경 : Node.js 고유의 API.

- 사용자 정의 객체 (user-defined objects)
  - 사용자가 직접 정의한 객체.

21.2 표준 빌트인 객체

- 생성자 함수 객체인 표준 빌트인 객체는 프로토타입 메서드와 정적 메서드를 제공.
- 생성자 함수 객체가 아닌 표준 빌트인 객체는 정적 메서드만 제공.

```
// String 생성자 함수에 의한 String 객체 생성
const strObj = new String('Lee'); // String {"Lee"}
console.log(typeof strObj);       // object

// Number 생성자 함수에 의한 Number 객체 생성
const numObj = new Number(123); // Number {123}
console.log(typeof numObj);     // object
```

```
// Boolean 생성자 함수에 의한 Boolean 객체 생성
const boolObj= new Boolean(true); // Boolean {true}
console.log(typeof boolObj);      // object

// Function 생성자 함수에 의한 Function 객체(함수) 생성
const func = new Function('x', 'return x * x'); // ƒ anonym
ous(x )
console.log(typeof func);                       // function

// Array 생성자 함수에 의한 Array 객체(배열) 생성
const arr = new Array(1, 2, 3); // (3) [1, 2, 3]
console.log(typeof arr);        // object

// RegExp 생성자 함수에 의한 RegExp 객체(정규 표현식) 생성
const regExp = new RegExp(/ab+c/i); // /ab+c/i
console.log(typeof regExp);         // object

// Date 생성자 함수에 의한 Date 객체 생성
const date = new Date();  // Fri May 08 2020 10:43:25 GMT+0
900 (대한민국 표준시)
console.log(typeof date); // object
```

- 생성자 함수인 표준 빌트인 객체가 생성한 인스턴스의 프로토타입은 **표준 빌트인 객체의 prototype 프로퍼티에 바인딩된 객체.**

```
// String 생성자 함수에 의한 String 객체 생성
const strObj = new String('Lee'); // String {"Lee"}

// String 생성자 함수를 통해 생성한 strObj 객체의 프로토타입은 Strin
g.prototype이다.
console.log(Object.getPrototypeOf(strObj) === String.protot
ype); // true
```

- 다양한 기능의 빌트인 프로토타입 메서드를 제공.

- 인스턴스 없이도 호출이 가능한 빌트인 정적 메서드를 제공.

```
// Number 생성자 함수에 의한 Number 객체 생성
const numObj = new Number(1.5); // Number {1.5}

// toFixed는 Number.prototype의 프로토타입 메서드다.
// Number.prototype.toFixed는 소수점 자리를 반올림하여 문자열로 반환한다.
console.log(numObj.toFixed()); // 2

// isInteger는 Number의 정적 메서드다.
// Number.isInteger는 인수가 정수(integer)인지 검사하여 그 결과를 Boolean으로 반환한다.
console.log(Number.isInteger(0.5)); // false
```

21.3 원시값과 래퍼 객체

```
const str = 'hello';

// 원시 타입인 문자열이 프로퍼티와 메서드를 갖고 있는 객체처럼 동작한다.
console.log(str.length); // 5
console.log(str.toUpperCase()); // HELLO
```

- 원시값을 객체처럼 사용하면 자바스크립트 엔진은 암묵적으로 연관된 객체를 생성한 후, 프로퍼티에 접근하거나 메서드를 호출하고, 다시 원시값으로 되돌린다.

> 💡 래퍼 객체 (wrapper object)
> : 문자열, 숫자, 불리언 값에 대해 객체처럼 접근하면 생성되는 임시 객체.

```
const str = 'hi';

// 원시 타입인 문자열이 래퍼 객체인 String 인스턴스로 변환된다.
```

```
console.log(str.length); // 2
console.log(str.toUpperCase()); // HI

// 래퍼 객체로 프로퍼티에 접근하거나 메서드를 호출한 후, 다시 원시값으로 되돌린다.
console.log(typeof str); // string
```

- 래퍼 객체의 처리가 종료되면 자바스크립트 엔진은 식별자가 원시값을 갖도록 되돌리고, 래퍼 객체는 가비지 컬렉션의 대상이 된다.

```
// ① 식별자 str은 문자열을 값으로 가지고 있다.
const str = 'hello';

// ② 식별자 str은 암묵적으로 생성된 래퍼 객체를 가리킨다.
// 식별자 str의 값 'hello'는 래퍼 객체의 [[StringData]] 내부 슬롯에 할당된다.
// 래퍼 객체에 name 프로퍼티가 동적 추가된다.
str.name = 'Lee';

// ③ 식별자 str은 다시 원래의 문자열, 즉 래퍼 객체의 [[StringData]] 내부 슬롯에 할당된 원시값을 갖는다.
// 이때 ②에서 생성된 래퍼 객체는 아무도 참조하지 않는 상태이므로 가비지 컬렉션의 대상이 된다.

// ④ 식별자 str은 새롭게 암묵적으로 생성된(②에서 생성된 래퍼 객체와는 다른) 래퍼 객체를 가리킨다.
// 새롭게 생성된 래퍼 객체에는 name 프로퍼티가 존재하지 않는다.
console.log(str.name); // undefined

// ⑤ 식별자 str은 다시 원래의 문자열, 즉 래퍼 객체의 [[StringData]] 내부 슬롯에 할당된 원시값을 갖는다.
// 이때 ④에서 생성된 래퍼 객체는 아무도 참조하지 않는 상태이므로 가비지 컬렉션의 대상이 된다.
console.log(typeof str, str);
```

```
const num = 1.5;

// 원시 타입인 숫자가 래퍼 객체인 Number 객체로 변환된다.
console.log(num.toFixed()); // 2

// 래퍼 객체로 프로퍼티에 접근하거나 메서드를 호출한 후, 다시 원시값으로 되돌린다.
console.log(typeof num, num); // number 1.5
```

- null, undefined는 래퍼 객체를 생성하지 않으므로, 객체처럼 사용하면 에러가 발생한다.

21.4 전역 객체

- 전역 객체 (global object)
  - 코드가 실행되기 이전 단계에 생성.
  - 자바스크립트 엔진에 의해 제일 먼저 생성되는 특수한 객체.
  - 어떤 객체에도 속하지 않은 최상위 객체.

- 브라우저 환경에서의 전역 객체 : window (self, this, frames)
- Node.js 환경에서의 전역 객체 : global

```
// 브라우저 환경
globalThis === this   // true
globalThis === window // true
globalThis === self   // true
globalThis === frames // true

// Node.js 환경(12.0.0 이상)
globalThis === this   // true
globalThis === global // true
```

전역 객체의 특징

- 개발자가 의도적으로 생성할 수 없다.
- 전역 객체를 생성할 수 있는 생성자 함수가 없다.
- 전역 객체의 프로퍼티를 참조할 때 window (또는 global)를 생략할 수 있다.

```
// 문자열 'F'를 16진수로 해석하여 10진수로 변환하여 반환한다.
window.parseInt('F', 16); // -> 15
// window.parseInt는 parseInt로 호출할 수 있다.
parseInt('F', 16); // -> 15

window.parseInt === parseInt; // -> true
```

- 전역 객체는 모든 표준 빌트인 객체를 프로퍼티로 갖고 있다.
- 자바스크립트 실행 환경에 따라 추가적으로 프로퍼티 및 메서드를 갖는다.
- var로 선언한 전역 변수, 암묵적 전역 그리고 전역 함수는 전역 객체의 프로퍼티가 된다.

```
// var 키워드로 선언한 전역 변수
var foo = 1;
console.log(window.foo); // 1

// 선언하지 않은 변수에 값을 암묵적 전역. bar는 전역 변수가 아니라
전역 객체의 프로퍼티다.
bar = 2; // window.bar = 2
console.log(window.bar); // 2

// 전역 함수
function baz() { return 3; }
console.log(window.baz()); // 3
```

- let, const 전역 변수는 전역 객체 프로퍼티가 아니다.

```
let foo = 123;
console.log(window.foo); // undefined
```

- 브라우저 환경의 모든 자바스크립트 코드는 하나의 전역 객체 window를 공유한다.

21.4.1 빌트인 전역 프로퍼티

- 전역 객체의 프로퍼티.
- 애플리케이션 전역에서 사용하는 값을 제공.

Infinity

무한대를 나타내는 숫자값 Infinity.

```
// 전역 프로퍼티는 window를 생략하고 참조할 수 있다.
console.log(window.Infinity === Infinity); // true

// 양의 무한대
console.log(3/0);  // Infinity
// 음의 무한대
console.log(-3/0); // -Infinity
// Infinity는 숫자값이다.
console.log(typeof Infinity); // number
```

NaN

숫자가 아님 (Not-a-Number)을 나타내는 숫자값 NaN.

```
console.log(window.NaN); // NaN

console.log(Number('xyz')); // NaN
console.log(1 * 'string');  // NaN
console.log(typeof NaN);    // number
```

undefined

원시 타입 undefined.

```
console.log(window.undefined); // undefined

var foo;
console.log(foo); // undefined
console.log(typeof undefined); // undefined
```

21.4.2 빌트인 전역 함수

- 애플리케이션 전역에서 호출할 수 있는 빌트인 함수.
- 전역 객체의 메서드.

eval

문자열을 인수로 전달받는다.

표현식이라면 런타임에 평가하여 값을 생성.

표현식이 아닌 문이라면 런타임에 실행.

```
// 표현식인 문
eval('1 + 2;'); // -> 3
// 표현식이 아닌 문
eval('var x = 5;'); // -> undefined

// eval 함수에 의해 런타임에 변수 선언문이 실행되어 x 변수가 선언되었
다.
console.log(x); // 5

// 객체 리터럴은 반드시 괄호로 둘러싼다.
const o = eval('({ a: 1 })');
console.log(o); // {a: 1}

// 함수 리터럴은 반드시 괄호로 둘러싼다.
const f = eval('(function() { return 1; })');
console.log(f()); // 1
```

- 여러 개의 문이라면 모든 문을 실행하고 마지막 결과값을 반환.

```
console.log(eval('1 + 2; 3 + 4;')); // 7
```

- eval 함수는 자신이 호출된 위치에 있는 기존 스코프를 런타임에 동적으로 수정.

```
const x = 1;

function foo() {
  // eval 함수는 런타임에 foo 함수의 스코프를 동적으로 수정한다.
```

```
  eval('var x = 2;');
  console.log(x); // 2
}

foo();
console.log(x); // 1
```

- eval 함수에 전달된 코드는 이미 그 위치에 존재하던 코드처럼 동작.
- 엄격 모드에서는 eval 함수의 자체적인 스코프가 생성.

```
const x = 1;

function foo() {
  'use strict';

  // strict mode에서 eval 함수는 기존의 스코프를 수정하지 않고 eval
함수 자신의 자체적인 스코프를 생성한다.
  eval('var x = 2; console.log(x);'); // 2
  console.log(x); // 1
}

foo();
console.log(x); // 1
```

- let, const 변수는 암묵적으로 엄격 모드 적용.

```
const x = 1;

function foo() {
  eval('var x = 2; console.log(x);'); // 2
  // let, const 키워드를 사용한 변수 선언문은 strict mode가 적용된
다.
  eval('const x = 3; console.log(x);'); // 3
  console.log(x); // 2
}
```

```
foo();
console.log(x); // 1
```

- eval 함수의 사용은 금지해야 한다.
  - 보안에 매우 취약.
  - 최적화가 되지 않으므로 코드 실행 및 처리 속도가 느리다.

isFinite

유한수이면 true, 무한수이면 false 반환.

```
// 인수가 유한수이면 true를 반환한다.
isFinite(0);    // -> true
isFinite(2e64); // -> true
isFinite('10'); // -> true: '10' → 10
isFinite(null); // -> true: null → 0

// 인수가 무한수 또는 NaN으로 평가되는 값이라면 false를 반환한다.
isFinite(Infinity);  // -> false
isFinite(-Infinity); // -> false

// 인수가 NaN으로 평가되는 값이라면 false를 반환한다.
isFinite(NaN);     // -> false
isFinite('Hello'); // -> false
isFinite('2005/12/12'); // -> false
```

`isFinite(null)`은 true. null을 숫자로 변환하면 0이기 때문.

```
console.log(+null); // 0
```

isNaN

NaN 여부를 검사하여 불리언으로 반환.

```
// 숫자
isNaN(NaN); // -> true
isNaN(10);  // -> false
```

```
// 문자열
isNaN('blabla'); // -> true: 'blabla' => NaN
isNaN('10');     // -> false: '10' => 10
isNaN('10.12');  // -> false: '10.12' => 10.12
isNaN('');       // -> false: '' => 0
isNaN(' ');      // -> false: ' ' => 0

// 불리언
isNaN(true); // -> false: true → 1
isNaN(null); // -> false: null → 0

// undefined
isNaN(undefined); // -> true: undefined => NaN

// 객체
isNaN({}); // -> true: {} => NaN

// date
isNaN(new Date());            // -> false: new Date() => Nu
mber
isNaN(new Date().toString()); // -> true:  String => NaN
```

parseFloat

실수로 해석하여 반환.

```
// 문자열을 실수로 해석하여 반환한다.
parseFloat('3.14');  // -> 3.14
parseFloat('10.00'); // -> 10

// 공백으로 구분된 문자열은 첫 번째 문자열만 변환한다.
parseFloat('34 45 66'); // -> 34
parseFloat('40 years'); // -> 40

// 첫 번째 문자열을 숫자로 변환할 수 없다면 NaN을 반환한다.
parseFloat('He was 40'); // -> NaN
```

```
// 앞뒤 공백은 무시된다.
parseFloat(' 60 '); // -> 60
```

parseInt

정수로 해석하여 반환.

```
// 문자열을 정수로 해석하여 반환한다.
parseInt('10');     // -> 10
parseInt('10.123'); // -> 10
```

```
parseInt(10);     // -> 10
parseInt(10.123); // -> 10
```

두 번째 인수로 진법을 나타내는 기수 (2 ~ 36)을 전달 가능.

```
// 10'을 10진수로 해석하고 그 결과를 10진수 정수로 반환한다
parseInt('10'); // -> 10
// '10'을 2진수로 해석하고 그 결과를 10진수 정수로 반환한다
parseInt('10', 2); // -> 2
// '10'을 8진수로 해석하고 그 결과를 10진수 정수로 반환한다
parseInt('10', 8); // -> 8
// '10'을 16진수로 해석하고 그 결과를 10진수 정수로 반환한다
parseInt('10', 16); // -> 16
```

반대로 10진수를 해당 기수의 문자열로 반환하려면 `Number.prototype.toString()` 메서드 사용.

```
const x = 15;

// 10진수 15를 2진수로 변환하여 그 결과를 문자열로 반환한다.
x.toString(2); // -> '1111'
// 문자열 '1111'을 2진수로 해석하고 그 결과를 10진수 정수로 반환한다
parseInt(x.toString(2), 2); // -> 15

// 10진수 15를 8진수로 변환하여 그 결과를 문자열로 반환한다.
x.toString(8); // -> '17'
// 문자열 '17'을 8진수로 해석하고 그 결과를 10진수 정수로 반환한다
```

```
parseInt(x.toString(8), 8); // -> 15

// 10진수 15를 16진수로 변환하여 그 결과를 문자열로 반환한다.
x.toString(16); // -> 'f'
// 문자열 'f'를 16진수로 해석하고 그 결과를 10진수 정수로 반환한다
parseInt(x.toString(8), 8); // -> 15

// 숫자값을 문자열로 변환한다.
x.toString(); // -> '15'
// 문자열 '15'를 10진수로 해석하고 그 결과를 10진수 정수로 반환한다
parseInt(x.toString()); // -> 15
```

```
// 16진수 리터럴 '0xf'를 16진수로 해석하고 10진수 정수로 그 결과를 반환한다.
parseInt('0xf'); // -> 15
// 위 코드와 같다.
parseInt('f', 16); // -> 15
```

```
// 2진수 리터럴(0b로 시작)은 제대로 해석하지 못한다. 0 이후가 무시된다.
parseInt('0b10'); // -> 0
// 8진수 리터럴(ES6에서 도입. 0o로 시작)은 제대로 해석하지 못한다. 0 이후가 무시된다.
parseInt('0o10'); // -> 0
```

```
// 문자열 '10'을 2진수로 해석한다.
parseInt('10', 2); // -> 2
// 문자열 '10'을 8진수로 해석한다.
parseInt('10', 8); // -> 8
```

```
// 'A'는 10진수로 해석할 수 없다.
parseInt('A0'); // -> NaN
// '2'는 2진수로 해석할 수 없다.
parseInt('20', 2); // -> NaN
```

```
// 10진수로 해석할 수 없는 'A' 이후의 문자는 모두 무시된다.
parseInt('1A0'); // -> 1
// 2진수로 해석할 수 없는 '2' 이후의 문자는 모두 무시된다.
parseInt('102', 2); // -> 2
// 8진수로 해석할 수 없는 '8' 이후의 문자는 모두 무시된다.
parseInt('58', 8); // -> 5
// 16진수로 해석할 수 없는 'G' 이후의 문자는 모두 무시된다.
parseInt('FG', 16); // -> 15
```

```
// 공백으로 구분된 문자열은 첫 번째 문자열만 변환한다.
parseInt('34 45 66'); // -> 34
parseInt('40 years'); // -> 40
// 첫 번째 문자열을 숫자로 변환할 수 없다면 NaN을 반환한다.
parseInt('He was 40'); // -> NaN
// 앞뒤 공백은 무시된다.
parseInt(' 60 '); // -> 60
```

encodeURI / decodeURI

- encodeURI 완전한 URI를 문자열로 전달 받아 이스케이프 처리를 위한 인코딩.
- 인코딩 URI의 문자들을 이스케이프 처리하는 과정.
- 이스케이프 처리 네트워크를 통해 정보를 공유할 때 어떤 시스템에서도 읽을 수 있도록 아스키 문자 셋으로 변환하는 것.
- decodeURI 인코딩된 URI를 인수로 전달받아 이스케이프 처리 이전으로 디코딩.

```
const uri = 'http://example.com?name=이웅모&job=programmer&t
eacher';

// encodeURI 함수는 완전한 URI를 전달받아 이스케이프 처리를 위해 인코
딩한다.
const enc = encodeURI(uri);
console.log(enc);
// http://example.com?name=%EC%9D%B4%EC%9B%85%EB%AA%A8&job=
programmer&teacher

// decodeURI 함수는 인코딩된 완전한 URI를 전달받아 이스케이프 처리 이
```

```
전으로 디코딩한다.
const dec = decodeURI(enc);
console.log(dec);
// http://example.com?name=이웅모&job=programmer&teacher
```

encodeURIComponent / decodeURIComponent

- encodeURIComponent URI 구성 요소를 인수로 전달받아 인코딩.
- decodeURIComponent URI 구성 요소를 디코딩.
- encodeURIComponent
  - 인수로 전달된 문자열을 URI의 구성 요소인 쿼리 스트링의 일부로 간주.
  - 쿼리 스트링 구분자로 사용되는 =, ?, & 까지 인코딩.
- encodeURI
  - 매개변수로 전달된 문자열을 완전한 URI 전체로 간주.
  - 쿼리 스트링 구분자로 사용되는 =, ?, &은 인코딩하지 않는다.

```
// URI의 쿼리 스트링
const uriComp = 'name=이웅모&job=programmer&teacher';

// encodeURIComponent 함수는 인수로 전달받은 문자열을 URI의 구성요
소인 쿼리 스트링의 일부로 간주한다.
// 따라서 쿼리 스트링 구분자로 사용되는 =, ?, &까지 인코딩한다.
let enc = encodeURIComponent(uriComp);
console.log(enc);
// name%3D%EC%9D%B4%EC%9B%85%EB%AA%A8%26job%3Dprogrammer%26
teacher

let dec = decodeURIComponent(enc);
console.log(dec);
// 이웅모&job=programmer&teacher

// encodeURI 함수는 인수로 전달받은 문자열을 완전한 URI로 간주한다.
// 따라서 쿼리 스트링 구분자로 사용되는 =, ?, &를 인코딩하지 않는다.
enc = encodeURI(uriComp);
console.log(enc);
// name=%EC%9D%B4%EC%9B%85%EB%AA%A8&job=programmer&teacher
```

```
dec = decodeURI(enc);
console.log(dec);
// name=이웅모&job=programmer&teacher
```

21.4.3 암묵적 전역

- 선언하지 않은 식별자에 값을 할당한 경우, 암묵적으로 전역 객체의 프로퍼티로 할당.
- 단지 전역 객체의 프로퍼티로 할당했기 때문에 호이스팅이 발생하지 않는다.

```
// 전역 변수 x는 호이스팅이 발생한다.
console.log(x); // undefined
// 전역 변수가 아니라 단지 전역 객체의 프로퍼티인 y는 호이스팅이 발생하지 않는다.
console.log(y); // ReferenceError: y is not defined

var x = 10; // 전역 변수

function foo () {
  // 선언하지 않은 식별자에 값을 할당
  y = 20; // window.y = 20;
}
foo();

// 선언하지 않은 식별자 y를 전역에서 참조할 수 있다.
console.log(x + y); // 30
```

```
var x = 10; // 전역 변수

function foo () {
  // 선언하지 않은 식별자에 값을 할당
  y = 20; // window.y = 20;
  console.log(x + y);
}

foo(); // 30
```

```
console.log(window.x); // 10
console.log(window.y); // 20

delete x; // 전역 변수는 삭제되지 않는다.
delete y; // 프로퍼티는 삭제된다.

console.log(window.x); // 10
console.log(window.y); // undefined
```